VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

SAX PHP XML Parser

SAX PHP XML Parser

2011 Bc. Jakub Matuš

Zadání diplomové práce

Prohlášení Studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Datum: 6.5.2011 **Podpis**: ……

Jakub Matuš

Poděkování

Chtěl bych poděkovat vedoucímu své diplomové práce Ing. Zbyněk Strnad za cenné rady a připomínky, které měly velmi pozitivní vliv na kvalitu vyvíjeného modulu a celé této práce.

Abstrakt

Cílem práce je najít ideální řešení pro zpracování obsáhlých XML souboru s relativně složitou strukturou v jazyku PHP. Obsahem práce je vytvoření parseru, který bude přistupovat k XML efektivně jak z hlediska času zpracování, tak z hlediska náročnosti na paměť a celkového zatížení hardware.

Poslední částí práce je porovnání s již vytvořenými přístupy k XML jako jsou DOM či SimpleXML a tím tedy celkově zhodnotit práci s XML v jazyku PHP.

Klíčová slova: PHP, XML, SAX, DOM, SimpleXML, XML Schéma, webové služby, internetové aplikace

Abstract

The aim is to find the ideal solution for handling large-scale XML file with a relatively complex structure in PHP. The research work is to create a parser that will access the XML as efficiently in terms of processing time, so both in terms of memory and the overall burden of hardware. The last section is compared with the already developed approaches to XML such as DOM or SimpleXML and therefore to assess the overall work with XML in PHP.

Keywords: PHP, XML, SAX, DOM, SimpleXML, XML Scheme, webové služby, internetové aplikace

Obsah

1 Úvod

Předkládaná diplomová práce nazývaná SAX PHP XML Parser je zaměřená na vytvoření vlastního XML parseru, který využívá sekvenční metodu zpracování SAX, implementovaného v jazyce PHP.

V dnešní době je XML nejpoužívanější přenosová struktura, která je využívána například v mnoha B2B systémech. Výhodou XML je jeho jednoduchost syntaxe, jak při vytváření XML dokumentů tak jejich následné zpracování. XML je výborný formát pro uchovávání dat jakéhokoliv typu. Například v B2B systémech typu „dodavatel – odběratel“ se používá pro přenos informací o produktech. Odběratel odebírá exportovaný soubor od dodavatele, který stahuje každý den pro aktuálnost dat. Problém na straně odběratele však nastává v případech, kdy přenášený dokument je příliš velký (desítky MB) a jeho zpracování je tak časově i paměťově velice náročné. V jazyce PHP je zpracování takovýchto souboru za pomocí nativních knihoven velice náročné, hlavně z hlediska paměťových nároků.

Cílem diplomové práce je tedy vytvořit vlastní XML parser, který bude využívat sekvenčního zpracování souboru (po částech) a tím se snažit nároky na paměť minimalizovat, popřípadě umožnit se souborem pracovat tak, aby byl zpracovatelný v několika krocích nebo po částech. Před začátkem samotného zpracování bude implementována možnost filtrace obsahu dané XML struktury. Ta bude nastavovat parametry pro parser, podle kterých se bude rozhodovat, která část dokumentu se má nebo nemá zpracovat (převést do objektu). Jeho výsledkem je tedy objekt nebo pole, který reprezentuje XML strukturu. V případě objektu je tak možno dále nakládat s XML strukturou za pomocí několika funkcí jako jsou (getParent, getChildren, getNext, apod.), které slouží pro jeho lepší manipulaci v další části skriptu (ukládání do databáze, verifikace dat, apod.). V druhé části modulu je řešena dotazování pomocí podmnožiny XPath na výstup typu pole, který tak bude umožňovat lepší práci s daty, které byly odfiltrovány z obsahu dokumentu. V poslední části dokumentu je aplikovaná možnost validace podle XML schématu.

Diplomová práce je strukturována do šesti obsahových kapitol, které se dále rozčleňují na podkapitoly druhé a třetí úrovně. První kapitola je určena seznámení s XML, s existujícími přístupy k XML a seznámení se základy XPath. Druhá kapitola je úvodem do problematiky řešení pomocí SAX a jeho důvody.

Třetí, obsahová kapitola popisuje na začátku hlavní strukturu celého implementovaného modulu XML parseru. V další části této hlavní kapitoly je popsána implementace a použití filtrace

XML obsahu. Druhá polovina této kapitoly se věnuje samotnému převodu XML do objektu. Na konci kapitoly je popsán i jiný typ výstupu než objekt, kterým je pole.

Čtvrtá obsahová kapitola se věnuje řešení XPath a jeho možnosti použití. Na začátku kapitoly je uvedeno seznámení s implementovanými typy os XPath a v druhé části je věnována deskripci samotné implementace a její zpracování výsledků.

V následné kapitole je popsána validace podle XML schématu, kde začátek je zaměřen na samotnou charakteristiku XSD a postup jak takové schéma vytvořit, v druhé části je uvedeno jeho řešení ve vytvořeném modulu.

Závěrečná kapitola je věnována konečnému zhodnocení vytvořeného modulu XML parseru s nativními knihovnami – SimpleXML a DOM. V první části je vylíčeno měření vzhledem k náročnosti na paměť a druhá část je naopak zaměřená na měření celkové rychlosti zpracování.

2 Popis nástrojů pro práci s XML v PHP

2.1 Jazyk PHP

PHP (Hypertext Preprocessor, původně Personal Home Page) je skriptovací programovací jazyk, určený především pro programování dynamických internetových stránek. Mnohdy se začleňuje přímo do skladby jazyka HTML, XHTML, což lze využít při tvorbě webových aplikací. PHP lze použít i k tvorbě konzolových a desktopových aplikací.

Při použití PHP pro dynamické stránky jsou skripty realizovány na straně serveru – k uživateli je přenášen až výsledek jejich aktivity (interpret PHP skriptu je možné volat pomocí příkazové řádky). Syntaxe jazyka PHP je inspirována několika programovacími jazyky (Perl, C, Pascal a Java). PHP je nezávislý na platformě, rozdíly v různých operačních systémech se omezují na několik OS-závislých funkcí a skripty lze většinou mezi nimi přenášet bez jakýchkoli úprav.

PHP podporuje mnoho knihoven pro různé účely - např. zpracování textu, grafiky, práci se soubory, přístup k většině databázových systémů (mj. MySQL, Oracle, PostgreSQL, MSSQL), podporu celé řady internetových protokolů (HTTP, SMTP, SNMP, FTP, IMAP, POP3, LDAP, aj.), nebo pro tuto diplomovou práci potřebnou podporu formátu XML.

PHP je vedle ASP.NET nebo Java jedním ze dvou nejrozšířenějších skriptovacích jazyků pro webové aplikace. Oblíbeným je díky své jednoduchosti použití, mnohočetné zásobě funkcí a tomu, že kombinuje vlastnosti více programovacích jazyků a nechává tak vývojáři částečnou svobodu v syntaxi. V kombinaci s operačním systémem Linux, databázovým systémem (obvykle MySQL nebo PostgreSQL) a webovým serverem Apache je často využíván k tvorbě webových aplikací. Pro tuto kombinaci se vžila zkratka LAMP – tedy spojení Linux, Apache, MySQL a PHP nebo Perl. V PHP jsou napsány i ty největší internetové projekty. [1]

2.2 XML

Extensible Markup Language (zkráceně XML, česky rozšiřitelný značkovací jazyk) je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Je zjednodušenou podobou staršího jazyka SGML. Umožňuje snadné vytváření konkrétních značkovacích jazyků (tzv. aplikací) pro různé účely a různé typy dat. Používá se pro serializaci dat, v čemž se podobají např. JSON či YAML. Zpracování XML je podporováno řadou nástrojů a programovacích jazyků.

Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů, u kterých popisuje strukturu z hlediska věcného obsahu jednotlivých částí, nezabývá se vzhledem. [2]

```
<book name=“Moje první knížka“>
    <isbn>AAA00</isbn>
    <price>1.0.1></price>
    <vat>5</vat>
    <description>Popisek první knížky</description>
</book>
```

Ukázka XML dokumentu

Prezentace dokumentu (vzhled) může být definována pomocí kaskádových stylů. Další možností zpracování je transformace do jiného typu dokumentu, nebo do jiné aplikace XML.

2.2.1 Struktura XML

XML rozlišuje malá a velká písmena (case-sensitive). Základem každého XML dokumentu je hlavička. Základním kódováním XML je UTF-8, pokud je potřeba změnit definici kódování, je nutné to přidat do hlavičky XML dokumentu.

```
<?xml version="1.0" encoding="zvolené kódování"?>
```

Hlavička XML dokumentu

V XML je nadefinováno několik základních pojmů:

1. Element

Základním prvkem jazyka XML je **element**. Element je tvořen dvěma tagy (počátečním a koncový). Tag je vnořen mezi lomené závorky (<, >). Každý element musí být řádně uzavřen (na rozdíl od HTML). Obsah elementu může být prázdný nebo obsahovat text, obsahovat několik **sub-elementů** (child – „potomek“), případně obě možnosti najednou. Pokud je element prázdný, může být uzavřen zkráceně. [4]

```
<muj_element>obsah elementu</muj_element>
```

Ukázka jednouchého XML elementu

```
<element>
  <sub_element>text sub elementu</sub_element>
</element>
```

Ukázka složeného elementu

```
<muj_element />
```

Ukázka zkráceného elementu

V tvorbě XML dokumentu, existuje zásadní pravidlo, elementy se nesmí za žádných okolností křížit. To znamená, že pokud je vytvářen element (např. cena), který je potomkem elementu kniha, nesmí uvnitř něj být ukončen element, který je vytvořen nad ním (kniha, viz příklad *ukázka chybně zapsaného elementu*).

```
<kniha>
        <cena>
</ kniha>
        </cena>
```

Ukázka chybně zapsaného elementu

Element musí začínat písmenem nebo podtržítkem, poté může následovat jakýkoliv znak abecedy a speciální znaky (číslice, dvojtečky, tečky, podtržítka, pomlčky). Element naopak nesmí obsahovat mezeru, zpětné lomítko (pokud se nejedná o ukončovací element) a hvězdičku.

2. Atribut

Další důležitý prvek XML je **atribut**. Atribut je meta-informace vztažená k danému elementu (podobně jako u HTML). Element může obsahovat několik atributu. Atribut se vepisuje za počáteční tag elementu, oddělen mezerou. Začíná názvem atributu a obsah je uvnitř uvozovek. Omezení atributu je obdobné jako u elementu. [4]

```
<element aa="1" bb="druhy"> text </element>
```

Ukázka elementů s atributy

3. Komentáře

Pro tvorbu XML lze využít i komentářů uvnitř dokumentů. Obdobně jak jsme zvyklí v programovacím jazyce, slouží jako popisná informace nebo doprovodný text, který nemá vzhledem k překladu XML žádný význam. Komentář má v XML následující zápis:

```
<!— obsah komentáře -->
```

Ukázka komentáře

Celý komentář je opět uzavřen pomocí lomených závorek obdobně jako u elementu. Obsah však začíná speciálním znakem vykřičník, ten je pak následován dvěma pomlčkami, dále následuje mezera a obsah komentáře. Komentář je ukončen dvěma pomlčkami. [4]

4. CDATA

V XML se vyskytuje spousta speciálních znaků. Jak již bylo v úvodu zmíněno, většina jich je povolena. Existuje ale i pár speciálních znaků, které se nesmí vyskytovat v textu či v obsahu elementu. Jedná se o znaky <, >, &. Tyto znaky se musí nahradit speciálními entitami (<, >, &). V obsahu elementů je však přeformátování tolika znaků zbytečné. Může být využito CDATA, pokud je potřeba v XML přenášet např. zdrojové kódy. Tyto speciální entity se vloží do CDATA a text uvnitř nebude nijak konvertován ani mít speciální význam pro XML strukturu. Syntaxe tedy je: *<![CDATA[obsah]]>*. [4]

5. Uzly

Uzel je pojem, který se bude často vyskytovat ve spojení se syntaxí XPath. Specifikace XPath definuje 7 druhů uzlů – elementy, atributy, texty, jmenné prostory (*namespace*), předpisy *(processing-instruction*), komentáře (*comment*) a uzly dokumentů (*root*). Všechny XML dokumenty jsou chápány jako stromy uzlů. Kořen takového stromu je nazýván uzlem dokumentu (tzv. kořenový uzel). [5]

2.2.2 Využití XML

XML slouží například jako přenosová struktura ve webových službách, mezi které patří například SOAP nebo RPC. Kde např. u SOAPu využívá k definici samotného fungování SOAP serveru a jeho metod a datových typu (WSDL). U obou webových služeb pak XML vystupuje jako návratová hodnota. Díky tomu tak muže být SOAP server napsán v programovacím jazyce PHP a na straně klienta, který tvoří požadavky na server a ty pak zpracovává, může být implementován v jiném jazyce, např. ASP.NET

Díky jeho nezávislosti na programovací platformě a široké použitelnosti se stalo i volbou v přenosu dat z internetových obchodů například na agregační servery jakými jsou Zbozi.cz, Heureka.cz, HledejCeny.cz. Každý obchod si tak vytvoří výstupní soubor v předdefinovaném formátu.

XML tak nahrazuje ve výše zmíněných webových službách a výměnných souborech mezi různými aplikacemi nepoužitelný formát *.csv.

2.3 Základní přístupy k zpracování XML v jazyce PHP

Základní přístupy k XML v PHP dělíme podle zpracování:

- *paměťové*
- *sekvenční*

V prvním případě je celá XML struktura nahrána do paměti a nad ní se pak vykonávají pomocí tříd jednotlivé úkony. Paměť v tomto případě představuje především velké omezení a tím i zpomalení celého průchodu aplikace XML strukturou.

V druhém případě je XML struktura zpracovávána po částech (sekvenčně). V paměti je tedy pouze část, která se zpracovává. Tím je tedy dosaženo efektivnějšího zacházení s pamětí, avšak s delší dobou zpracování.

Hlavní rozdíl je tedy v práci s pamětí, která je velice důležitým faktorem pro rychlost dané aplikace a celkového zpracování XML. Výhodou sekvenčního zpracování je tedy úspornost paměti. Jeho nevýhodou je však složitost implementace samotného zpracování XML a tím i větší náročnost na dobu zpracování. Jednoduchost se tedy stává výhodou paměťového zpracování, kdy implementaci zpracování není nutné prakticky řešit. To samozřejmě usnadňuje především použití takového zpracování pro programátory začátečníky.

PHP obsahuje již tří základní knihovny pro přístup k XML. První je SimpleXML, druhou variantou je DOM, obě varianty pracují s prvním typem zpracováním (paměťovým). Třetí knihovna pro přístup k XML v PHP je SAX. Každá podkapitola (2.3.1 a 2.3.2) je věnována jednotlivým přístupům. V následné kapitole (2.4) jsou shrnuty jednotlivé přístupy a jejich výhody a nevýhody, případně rozdíly. V poslední kapitole (2.5) jsou zmíněny i úplně jiné přístupy k XML v PHP a jejich problematika. Knihovna SAX je rozepsána zvlášť v kapitole č. 3.

2.3.1 DOM (Document Object Model)

Jedná se o jednu z prvních tříd pro práci s XML v jazyce PHP. Na rozdíl od SimpleXML umí kromě parsování XML struktury, XML strukturu i vytvářet.

Tento přístup není znám určitě pouze v jazyce PHP. Používá se už dostatečně dlouho i v ostatních programovacích jazycích.

DOM reprezentuje schéma XML jako strom jednotlivých prvků. Parsování dokumentu není úplně zcela ideální. DOM totiž celou XML strukturu nahrává do paměti, bez možnosti omezení prvků či limitu. Převádění struktury však lze rozdělit do několika dalších podtříd, kdy např. prvek elementu

můžeme reprezentovat vlastní třídou *DOMNode*, tak samo i samotný element (uzel) lze reprezentovat jako podtřída *DOMElement*.

Jelikož tato třída se objevuje už od verze PHP 4 a nabízí spoustu možnosti čtení, převádění a jiných prostředku pro usnadnění a zpřehlednění práce s touto třídou. Např. načítání z HTML struktury, validace.

Stejně jako druhá jmenována třída má DOM svou konverzi z SimpleXML za pomocí metody **dom_import_simplexml**().

2.3.2 SimpleXML

Objevuje se od verze PHP 5. Jedná se o velmi jednoduchý a rychlý přístup k XML, který zpracovává soubor XML do objektu. Tento princip je částečně i základem pro tuto práci.

SimpleXML převádí prvky na atributy SimpleXML objektu. Když se v takovém prvku objeví více prvků na jedné úrovni takovéhoto prvků, je převeden do asociativního pole. Data (textový údaj prvku) jsou převedeny do řetězce (string). Pokud má více než jeden textový uzel, budou uspořádány v pořadí, v jakém se nacházejí.

Tato metoda tedy převádí strukturu do objektu, které jsou uloženy v paměti. Jednotlivé dílčí pod elementy jsou jako další pod objekt. Jedná se tedy o objektově orientovaný přístup k XML struktuře.

Z jednoduché struktury zachycené v kapitole 2.2, je možno se snadno dotázat na popisek knížky. Viz ukázka kódu níže.

```
$xml = simplexml_load_string($struktura_xml);

echo $xml->description; // Vypíše se "Popisek první knížky"
```

Příklad použití SimpleXML

Samotné načítání XML se dá provádět třemi rychlými metodami.

- **simplexml_load_file**() – načítáme XML strukturu ze souboru
- **simplexml_load_string** () – funkce načítá/převádí XML z textového řetězce, např. vytvoříme v průběhu kódu, jednoduchou strukturu XML do stringu, kterou pak načteme díky této metodě do SimpleXML objektu.
- **simplexml_import_dom**() – převádění z DOM struktury

2.4 Rozdíly a nevýhody mezi přístupy SimpleXML a DOM

Hlavním rozdílem je práce s paměti. DOM celou strukturu stále drží v paměti a přistupuje k ní, u SimpleXML se to využívá k prvotní inicializaci, ale pak je v paměti uložena objektová struktura dokumentu, která má přesně definované parametry.

Nevýhody přístupu u SimpleXML jsou malé možnosti třídy k postupnému procházení elementů a sub-elementů.

Z nevýhod třídy DOM logicky vyplývá, že DOM je naprosto nepostačující pro velké soubory dat např. z B2B (Business-to-business) systému které posílají XML soubory o velikosti několika MB. Toto se samozřejmě dá obejít navýšením paměti pro cache Apache serveru, to samozřejmě nejde do nekonečna a práce se tím i zpomaluje a hlavně zatěžuje server.

Proto se DOM používá pro některé části dokumentu s pevně daným rozsahem nebo právě pro jeho druhou stránku a tou je generování struktury.

Tedy pokud je potřeba široká škála možnosti s danou strukturou, zvolí se přístup DOM, avšak pouze pro malé struktury.

Pokud se hledá jednoduchost a rychlost, zvolí se přístup SimpleXML. Obě metody nejsou vhodnými kandidáty s rozsáhlou strukturou a velkým obsahem dat, kvůli paměti.

2.5 Jiné přístupy

Výše zmíněné přístupy samozřejmě nejsou jediné k XML, které se v programovacím jazyce PHP dají využít. Jedná se však o přístupy, které jsou implementovány jako rozšiřující knihovny základního PHP. Jestliže chcete využívat tyto knihovny, tak musí být nainstalovány přímo na serveru, což vyžaduje administrátorský zásah.

Tyto možnosti se tak mohou používat na vlastních serverech nebo těch, na kterých jsou tyto knihovny doinstalovány / povoleny. Neboť se jedná o funkce, které nejsou primárně určené pro jazyk PHP a jsou jakýmisi modifikátory pro něj.

Proto jsou tyto funkce jako obsah firemní aplikace, která je musí využívat prakticky nepoužitelné.

2.5.1 Saxon

Jednou z těchto funkcí může být SAXON. Jedná se o XSLT procesor, primárně určen pro jazyk Java, který se objevuje jako dostupná modifikace pro PHP. Tato modifikace však vyžaduje daleko více

instalačních modulu na serveru k běhu tohoto přístupu. Jinak řečeno, je nutné na cílovém prostředí (serveru) instalovat potřebné knihovny pro SAXON. Proto jako řešení, které by nefungovalo nativně na všech serverech, je nepoužitelné.

2.5.2 AltovaXML

AltovaXML je obdobným přístupem jako prvně zmíněný SAXON. Opět se jedná o XSLT procesor, který vyžaduje speciální instalaci na serveru.

2.6 XPath

XPath je důležitý bod pro práci s jazykem XML. Jedná se o dotazovací jazyk, kterým se zadává určitý výraz (dotaz, cestu) k jednotlivým elementům. Aby bylo možno vytvářet různé výrazy, je třeba znát strukturu dokumentu, nad kterým má být výraz proveden.

Výsledkem výrazu bývá nejčastěji množina uzlů. XPath si převede XML strukturu na stromovou strukturu, která je tvořena jednotlivými uzly.

Jeho výrazy často připomínají URL cesty, cestu k adresářové struktuře na disku nebo jazyk SQL. XPath je s jazykem XML využíván velice často a pochopení jeho syntaxe není zcela triviální.

2.6.1 Syntaxe XPath

XPath výraz je složen z několika jednotlivých kroků (úrovní), které jsou odděleny lomítkem /. Každý z těchto kroků se skládá z:

- identifikátoru osy (primárně ::child osy)
- test uzlu
- podmínky (predikátu)

Jako povinný je však pouze *test uzlu*. Celé vyhodnocení výrazu se provádí zleva doprava. Každý výraz se aplikuje na výsledek předešlého výrazu, při počátku se vychází z aktuálního elementu dokumentu.

Celý postup začíná vyhodnocením **identifikátoru osy** (nejčastěji child osy, jedná se o podřízené uzly aktuálního uzlu), po té následuje vymezení množiny uzlu podle **testu uzlu** (např. název elementu). Nakonec se aplikují podmínky. Uzly, které vyhovují, tak postupují jako základ pro další krok výrazu.

Podmínky, které se zadávají v hranatých závorkách, mohou využívat jakékoliv operátory pro syntaxi podmínek např. (<, >, =, *, - , div, mod, and, or a jiné).

2.6.2 Osy XPath

Osa určuje vztah v rámci stromu mezi aktuálním kontextovým uzlem a uzly vybíranými. V XPath se vyskytuje 13 identifikátorů os:

1. **child::** - přímí potomci aktuálního uzlu.
2. **descendand::** - všichni potomci aktuálního uzlu.
3. **descendand-or-self::** - aktuální uzel a všichni potomci.
4. **self::** - aktuální uzel.
5. **ancestor-or-self::** - aktuální uzel a všichni jeho předci.
6. **ancestor::** - všichni předci aktuálního uzlu.
7. **parent::** - rodič aktuálního uzlu.
8. **following::** - všechny uzly, které se v toku XML dokumentu nacházejí za aktuálním uzlem.
9. **preceding::** - všechny uzly, které se v toku XML dokumentu nacházejí před aktuálním uzlem.
10. **following-sibling::** - všichni následující sourozenci aktuálního uzlu.
11. **preceding-sibling::** - všichni předcházející sourozenci aktuálního uzlu.
12. **attribute::** - atributy aktuálního uzlu.
13. **namespace::** - deklarované jmenné prostory

2.6.3 Příklady XPath

Pro snadnější pochopení jazyka XPath je uvedeno několik příkladů.

Tab. 1 Ukázky XPath výrazů

Výraz	Popis výrazu
/*	kořenový uzel včetně všech potomků
//cena	element cena, kdekoliv
/produkt/cena	výběr všech elementů *cena,* který je přímým potomkem elementu *produkt*
/produkty/produkt/*	výběr všech produktu, které mají přímého předka produkty
/produkt[cena>=500]	výběr všech elementů *produkt,* který má potomka *cena* a jeho hodnota je větší nebo rovna 500
/produkt/cena[1]	výběr prvního elementů *cena,* který má předka produkt
/produkt[@vyrobce]	výběr všech produktů, které mají atribut výrobce
/produkty/produkt/.[:last]	poslední element produkt

2.7 Charakteristika XSD

Tato kapitola je zaměřená na charakteristiku XML schématu, jak jej nadefinovat a jeho základní prvky, které se v něm mohou objevovat. Základem je tedy vytvoření souboru, který bude obsahovat dané schéma. Soubor musí mít koncovku ***.xsd**

```
<?xml version="1.0" encoding="UTF-8"?>
```

Hlavička XML dokumentu

Na začátku dokumentu je třeba opět definovat hlavičku s možnosti nastavením kódováním. Celé schéma se pak definuje do hlavního elementu **xs:schema**.

```
<xs:schema>
    <!--obsah schématu-->
</xs:schema>
```

Ukázka hlavní struktury XSD dokumentu

Obsahem schéma jsou pak definice jednotlivých elementu a předdefinovaných typů. Každý element obsahuje pár základních definic (prvků). Element musí mít jméno, dále je definován jeho typ (Simple/Complex Type), sekvence jeho potomků a nastavení atributů. To samozřejmě není veškeré definování u jednotlivého elementu, jedná se o základní výčet možností. [6]

Mnohem srozumitelnější bude interpretace na příkladu. Mějme následující XML strukturu:

```
<kniha nazev="První kniha">
  <cena>500</cena>
  <dph>20</dph>
  <popis>Nová kniha spisovatele...</popis>
</kniha>
```

Příklad XML struktury

XML schéma pak bude vypadat takto:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="kniha">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="cena" type="xs:float"></xs:element>
      <xs:element name="dph" type="xs:integer"></xs:element>
      <xs:element name="popis" type="xs:string"></xs:element>
    </xs:sequence>
    <xs:attribute name="nazev" type="xs:string"></xs:attribute>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML schéma

Na XS je vidět, že atributy se definují až za sekvencí potomku a každý element musí mít pevně daný typ. Jak už bylo zmíněno, typy se mohou předefinovat na začátku XS dokumentu. Jedná se např. opakující se strukturu, nebo typ, který není primitivní, ale je složen z další sekvence elementů. Definice vlastních typu slouží hlavně pro přehlednost XS schématu.

3 Analýza řešení s využitím SAX

3.1 SAX (Simple API for XML)

SAX (Simple API for XML – český „jednoduchý přístup k XML“). Jedná se o sekvenční zpracování XML, kde se soubor (obsah XML) rozdělí na několik jednotlivých části (počáteční a koncové prvky, obsahy, atributy, komentáře atd.). Postupně se pak volají jednotlivé události, které ohlašují nalezení konkrétní části. Způsob zpracování těchto jednotlivých událostí je pak již plně v kompetenci programátora. [3]

K dokumentu tedy nepřistupujeme náhodně ale sekvenčně. Čímž vyčnívá jeho obrovská výhoda a tou je nízká náročnost na paměť. Samotné zpracování však u SAX bývá pomalejší než u SimpleXML nebo DOM.

Jedná se o nejstarší knihovnu pro zpracování XML v jazyce PHP. Prvotně byla implementována do programovacího jazyku Java. Následně se však i objevovala v dalších programovacích jazycích.

3.2 Proč pomocí SAX

Prvotní důvodem vzniku tohoto tématu diplomové práce byla, neschopnost zpracování velkých dat v XML za pomocí knihoven DOM a SimpleXML. SAX je ideálním pomocníkem, kterým můžeme zvolit svou cestu, která se zoptimalizuje pro vlastní potřeby. Ať už z hlediska rychlosti běhu skriptu nebo paměťové náročnosti.

Hlavní výhodou tedy budou minimální nároky na paměť. SAX tedy poslouží jako „dělič“ XML na jednotlivé části, které budou postupně zpracovávaný, dle nastavených kritérii.

3.3 Využití s modelem DOM

SAX sám o sobě nebude jediný, který bude využit k tvorbě vlastního přístupu. Ten bude nutný zkombinovat s vytvořeným objektem (podobnému objektu DOM) a to hlavně díky možnosti využití XPath a jiných objektových přístupu. Soubor XML tedy bude postupně čten za pomocí SAXu, který bude rozdělovat soubor do několika části, které budou převedeny do vlastního vytvořeného objektu.

Bylo by zbytečné takto rozdělovat celý dokument. V praxi nastane příklad, kdy soubor z jistého B2B sytému bude obsahovat 20tis. produktů. Ten se samozřejmě nezvládne importovat celý najednou. Díky SAXu se může tedy převést pouze část struktury do vytvořeného objektu. Kdy produkt např. obsahuje atributy (cena, popis, obrázek, marže). Dle zadaných kritérii lze vytáhnout pouze cenu a marži u všech produktů, tím se zredukuje celý obsah a náročnost na paměť.

4 Vytvoření objektového model pro práci s XML parserem na základě SAX

4.1 Struktura celého modulu XML SAX parseru

Vytvoření XML parseru založeného na SAXu lze chápat jako modul (komponentu). Ten byl rozdělen do několika tříd a souboru. Nejdůležitější třídou je *MySax*, která obsahuje metody (funkce) k obsluze XML parseru. Celý modul je tedy tvořen:

class ***MySax***	Hlavní třída XML parseru, obsahuje metody pro definování SAX funkci, filtrace a spojení výstupních objektů, XPath procesor a převod výstupního objektového modelu na pole.
class ***Element***	Třída, která zastupuje každý jednotlivý element z XML struktury.
class ***Level***	Zastupuje jednotlivý krok ve filtraci pro XML parser.
class ***Filter***	Třída, která vyhodnocuje nastavení cesty pro filtraci na daném prvku, který načte XML parser.
class ***SchemaValidator***	Tato třída provádí filtraci na vstupním XML, za pomocí XML schéma a využití nativní třídy XML Reader.

Základní částí vytvořeného XML parseru, který je předmětem této diplomové práce, bylo převést XML do objektového modelu připomínající DOM model na základě SAX.

Parser byl rozdělen na tři základní kroky:

1. nastavení filtrace XML obsahu
2. převod do objektového modelu
3. sloučení výsledků filtraci do jednoho objektového modelu

4.2 Filtrace XML obsahu

Filtrace je součástí převodu do objektového modelu. Její nastavení se však určuje před spuštění samotného parseru. Nastavení filtrace si lze představit jako cestu XPath, která má však mnohem snadnější definici a užívání.

Filtrace slouží k oddělení nepotřebných částí v XML od částí, nad kterými je třeba dále pracovat. Představme si např. XML soubor, který obsahuje seznam produktů, seznam výrobců a údaje o dostupnosti. V kroku kdy je potřeba v aplikaci nahrát seznam produktů, nebude potřeba seznam o výrobcích ani údaje o dostupnosti. Proto bude nastavena filtrace, která nám zpracuje pouze část XML a ta vyhovuje nastavené cestě ve filtraci.

Díky filtraci se tak sníží zpracovávaný obsah v paměti již při vstupu a tím se zrychlí průchod ve vytvořeném objektu.

V následující části budou popsány možnosti nastavení filtrace a její principy v zpracování SAX parseru.

4.2.1 Možnosti cest filtrace

Jak už bylo napsáno v úvodu této kapitoly, nastavení filtrace připomíná syntaxi XPath, je to však pouze malá část. Stejně jako u XPath je cesta filtrace rozdělena na jednotlivé úrovně za pomocí lomítek. Každá úroveň je složena z:

- názvu elementu
- omezení potomků (childs)
- omezení atributu (attributes)

```
/books/book{price}[name]
```

Možnost zadání filtrace

Tento příklad vybere všechny knížky a pouze s potomkem *price* a atributem *name*. U názvu elementu můžeme použít *:first*, který vybere pouze první element, ke kterému tuto funkci přiřadíme. Např. */book:first* – vrátí do objektu pouze první element knížky.

4.2.2 Nastavení filtrace

Filtrace se nastavuje za pomocí metody *set_filtration($paths = array ()), která* je v třídě *MySax*.

Vstupem této funkce je pole jednotlivých filtrací, které jsou zadány jako řetězec. Je tedy možné zadat při jednom zpracování více cest filtrací najednou.

A to hlavně z důvodu, že XML dokument nemusí mít vždy pouze jeden hlavní element (v příkladu výše obsahuje seznam produktů, seznam výrobců a seznam dostupností). Proto můžeme nastavit filtraci zvlášť na každý hlavní element (produkty, výrobci a dostupnosti).

```
$paths = array(
        '/produkty/produkt/cena',
        '/vyrobci/vyrobce/id,
        '/dostupnosti/dostupnost/nazev
);
```

Nastavení filtrace

Výstupem parseru při takovémto filtru bude objekt obsahující seznam produktů a pouze jejich elementu *cena*, seznam id jednotlivých výrobců a názvy všech dostupností.

Ve funkci *set_filtration()* se tedy postupně prochází vstupní pole. Každá cesta (jeden prvek vstupního pole) je poté inicializována jako třída *Filter*.

```
foreach($paths as $path){
        if (strpos($path, '/') === false) {
            return false;
          }
        $this->filter[] = new Filter($path);
}
```

Ukázka inicializace filtrů

Ve třídě *Filter* je každá cesta rozparsována na jednotlivé úrovně dle dělícího znaku lomítka na třídy (*Level*) za pomocí funkce *Filter::parser($path)*.

```
$tokens = array_slice(preg_split('/(?<!=)\/(?![a-z-\s]*\])/', $this->path), 1);
foreach($tokens as $level){
        list($name,$first,$childs,$attributes) = self::parser($level);
        $this->levels[] = new Level($name,$first,$attributes);
 }
```

Rozdělení cesty na jednotlivé úrovně (Levels)

Třída Filter po zpracování konstruktoru obsahuje seznam jednotlivých úrovní a jejich nastavení pro element, atributy a potomky.

Výsledkem části nastavené filtrace obsahu je seznam filtrů, který bude využívat SAX při převodu do objektového modelu.

4.3 Převod do objektového modelu

Tato pasáž je věnována části, která popisuje, jak vytvořený modul převádí XML strukturu do zvoleného objektového modelu. SAX je v této diplomové práci prezentován nativní třídou v PHP, **XML Parser**.

4.3.1 Inicializace XML parseru

Základem XML parseru je nadefinovat jednotlivé funkce které budou parserem volány podle toho na jaký typ elementu narazí. Tato inicializace je tedy vytvořena hned v konstruktoru třídy ***MySax***.

```
$this->parser = xml_parser_create();
```

Inicializace XML parseru

Na začátku je vytvořena instance XML parseru. Po té je nastavena funkce pro počáteční a koncový tag jednotlivého elementu.

```
xml_set_element_handler(
    $this->parser,
     array($this,"onStartElement"), array($this,"onEndElement")
);
```

Definice funkcí pro počáteční a koncový tag

Na ukázce výše je vidět, že jako první parametr je proměnná parseru a druhým parametrem je pole, ve kterém jsou definovány názvy funkcí pro počáteční (první prvek pole) a koncový tag.

```
xml_set_character_data_handler(
  $this->parser,
  array($this,"onContent")
);
```

Nastavení názvu funkce pro datovou část elementu

Jako poslední krok je nutno nadefinovat funkci pro data elementu (obsah), opět jako vstup dvou parametrů (instance parseru, název funkce pro uložení obsahu elementu).

4.3.2 Zpracování XML dokumentu

Zpracování XML dokumentu je zahájena vyvoláním funkce *loadFromLocalFile* ze třídy ***MySax***. Jako vstup je zadáno umístění XML dokumentu. Zpracování celého XML dokumentu probíhá sekvenčně. XML soubor tedy čteme za pomocí funkce *fread*, řádek po řádku. Každý takový řádek je předáván funkci *xml_parse()*.

```
while ($x = fread($fp, 4096)){
    xml_parse($this->parser, $x, feof($fp)))
}
```

Postupné procházení souboru a volání paseru

Jako vstup (podobně jako u ostatních funkcí parseru) jsou: instance parseru, obsah řádku z funkce *fread* a finální data (*boolean*). Funkce xml_parse, vyvolává automaticky funkce, které byly definovány na začátku podle toho, kterou část elementu přečte (počátek, obsah, konec).

onStartElement()

Je funkce pro počáteční tag elementu. XML parser nám přesně definuje vstupní parametry, které předá do námi zvolené funkce při inicializaci. Parametry jsou tři:

1. instance parseru
2. název elementu
3. atributy elementu

Hlavní úkolem této funkce je projít nastavené filtry a zkontrolovat zda daný název elementu vyhovuje nastavenému filtru na dané úrovní (levelu), ve kterém se nachází.

```
$nodeRes = $filter->_do(
  strtolower($elementName),
  $this->level,
  $attributes
);
```

Funkce _do třídy Filter

V případě, že element je ve filtru funkcí *_do,* vyhodnocen jako žádoucí, tak se ukládá do proměnné *$filter->add_new* 1 a do *$filter->now_object* se přidá aktuální přečtený element jako objekt třídy Element. Jako parent (předek) toho elementu je uložen předchozí objekt uložený v *$filter->now_object*. V případě že funkce *_do* vrací false, ukládáme do proměnné *$filter->add_new* 0.

Proměnná *$filter->add_new* funguje jako zásobník, kdy nastavujeme sekvenci elementů, které máme a nemáme přidávat. Tento zásobník pak zpracovává poslední funkce *onEndElement()*.

Při každém přečtení této funkce je zvyšována proměnná třídy ***MySax*** *$level* o 1. To proto, aby modul věděl, na které úrovní se tento prvek nachází a tím lépe ho porovnával s filtry.

Každý filtr má parametr *$run*. Ten podává informaci (true/false) o tom, zda daný filtr je aktivní (má-li se jim prvek testovat) či nikoliv.

Na konci funkce je zvyšována proměnná třídy *MySax*, *$uniqueIdElementCounter* o jedničku. Každý element XML dokumentu má tak vygenerovaný unikátní klíč, který pak slouží pro sloučení objektů.

onContent()

Je funkce pro obsah elementu. Jedná se o čistě datovou část elementu nikoliv obsah celý (bez sub-elementů). Tato funkce je vždy volána ihned po *onStartElement*. To znamená, že pokud XML dokument obsahuje sekvenci elementů:

```
<kniha>
   <text_maly>text knihy</ text_maly >
   <text_velky>TEXT KNIHY</ text_velky >
</ kniha >
```

Příklad XML dokumentu

je sekvence volaných SAX funkcí následovná:

onStartElementu(kniha), onConten(kniha), onStartElementu(text_maly), onContent(text_maly), onEndElement(text_maly), onStartElementu(text_velky), onContent(text_velky), onEndElement(text_velky), onEndElemen(kniha).

Ukázka sekvence volaných funkcí XML parserem

Funkce přebírá opět přesně definované vstupní parametry (instance parseru, obsah elementu).

onEndElement()

Je funkcí pro koncový tag elementu. Vstupní parametry jsou obdobné jako při funkci *onStartElement*, bez posledního parametru atributu.

V této části dochází k samotnému ukládání do jednotlivého filtru a jejich proměnné myObject. Zde je využito funkce z třídy Element::set_child(), kde každý přidávaný element je přidán jako potomek (child) prvku, který je nastaven jako rodič (parent).

To jestli se prvek přidá či nikoliv, se vyhodnocuje z již zmíněného zásobníku *add_new*, kde se vždy vybere poslední přidaný prvek.

Na konci funkce, je opět snížena proměnná třídy *MySax*, *$level* o jedničku.

4.4 Sloučení výsledků filtrací

Výstupem po zpracování XML parserem, je nyní pole filtrů. Každý filtr si drží svůj objektový model. Je však zbytečné abychom vraceli pole objektových modelů, které by měli ve většině případů hodně podobnou strukturu. Proto na konci zpracování parseru, jsou tyto objekty sloučeny do jednoho.

Na to je použita funkci *MySax::merge*. Vstupem pro tuto funkci bude pole jednotlivých filtrů. Pokud je počet filtru menší než 2, bude vrácen automaticky první filtr jako výstupní objektový model.

Pokud je počet filtrů ≥ 2 slučujeme filtry v cyklu po dvou. Objekt z prvního filtru uložíme do *$tmp* proměnné. Po té jsou procházeny filtry v cyklu *for* a každý následující objektový model z filtru je spojen s objektem uloženým v proměnné *$tmp*. Výsledek je opět uložen do proměnné *$tmp*.

```
$temp = $filters_array[0];
for($i = 1; $i < count($filters_array); $i++){
    $temp->myObject = self::mergeObject(
        $temp->myObject,
        $filters_array[$i]->myObject
    );
    unset($filters_array[$i]);
}
return $temp;
```

Ukázka sloučení výsledků jednotlivých filtrů

Jednotlivé sloučení dvou objektu provádí funkce *MySax::mergeObject*. Ta páruje elementy podle jejich unikátních klíčů. Ten je uložen u každého elementu v proměnné *$_uniqueId*. Pokud tedy funkce najde dva elementy se stejným *$_uniqueId*, tak posléze spojí jejich potomky. Rekurzivně pak funguje i spojení potomků. Elementy, které se neobjevují v prvním objektu ale pouze v druhém, jsou přeneseny ihned. Výstupem je tedy ve finále pouze jeden objektový model.

```
<aaa pocet_prvku="2">
  <bbb>Text elementu bbb</bbb>
  <ccc>Text elementu ccc</ccc>
</aaa>
```

XML struktura

```
$paths = array( '/aaa/bbb','/aaa/ccc');
```

Nastavení filtrace

```
Element Object
 (
    [_level] => 0
    [_uniqueId] => 0
    [_name] => aaa
    [_child] => Array
       (
          [1] => Element Object
             (
                [_level] => 1
                [_uniqueId] => 1
                [_name] => bbb
                [_child] => Array()
                [_attributes] => Array()
                [_content] => Text elementu bbb
             )
       )
    [_attributes] => Array([POCET_PRVKU] => 2)
    [_content] =>
 )
```

```
Element Object
  (
     [_level] => 0
     [_uniqueId] => 0
     [_name] => aaa
     [_child] => Array
        (
           [2] => Element Object
              (
                 [_level] => 1
                 [_uniqueId] => 2
                 [_name] => ccc
                 [_child] => Array()
                 [_attributes] => Array()
                 [_content] => Text elementu ccc
              )
        )
     [_attributes] => Array([POCET_PRVKU] => 2)
     [_content] =>
  )
```

Objektový model prvního filtru (vlevo) a druhého filtru (vpravo)

```
Element Object
        (
          [_level] => 0
          [_uniqueId] => 0
          [_name] => aaa
          [_child] => Array
            (
              [1] => Element Object
                (
                  [_level] => 1
                  [_uniqueId] => 1
                  [_name] => bbb
                  [_child] => Array()
                  [_attributes] => Array()
                  [_content] => Text elementu bbb
                )
              [2] => Element Object
                (
                  [_level] => 1
                  [_uniqueId] => 2
                  [_name] => ccc
                  [_child] => Array()
                  [_attributes] => Array()
                  [_content] => Text elementu ccc
                )
            )
          [_attributes] => Array([POCET_PRVKU] => 2)
          [_content] =>

        )
```

Objektový model po sloučení filtrů

4.5 Jiný typ výstupu

XML parser má jako návratovou hodnotu objektový model, který vytvoří. V parseru bylo však nutno implementovat i jiný výstup, respektive možnost převodu a to na typ **pole**.

Pole je velice využívaný typ v jazyce PHP. Jelikož PHP není až tak striktní objektový programovací jazyk, **pole** proto ve většině případů nahrazuje mnohočetné vstupní parametry funkcí, ukládání dat (náhrada za List) apod.

Proto třída *MySax*, obsahuje funkci *to_array*. Jedná se o funkci, která převádí objektový model na **asociativní pole**. Toto pole má však přesně definovanou strukturu, protože musí obsahovat všechny informace jako objektový model. Každý prvek pole je tvořen ze tří částí:

- atributy daného elementu
- obsah
- potomci

Jako klíč v poli je uložen název elementu. Jako hodnota bude uloženo pole. To bude obsahovat jednotlivé elementy se stejným názvem na dané úrovní. Výstupem je pak pole:

```
$return = array(
   'aaa' => array(
      0 => array( '__attributes' => array('pocet_prvku'=>2), '__content' => null,
         bbb=> array( 0 => array( '__attributes' => array(), '__content' => 'Text elementu bbb' ) ),
        ccc=> array(  0 => array(  '__attributes' => array(), '__content' => 'Text elementu ccc' )  )
      )
   ));
```

Výstup jako pole, z přechozího příkladu po převodu z objektového modelu

5 XPath

Úvodu do problematiky XPath a její syntaxe, byla věnována druhá kapitola. V této kapitole je popsáno XPath řešení pro výstup z XML parseru.

V předcházející kapitole bylo uvedeno jako jiná možnost výstupu **pole**. Jedná se o mnohem jednodušší implementaci a částečně i rychlejší zpracování výsledků. Hlavním důvodem je však použitelnost pole v jazyce PHP, které bylo zmíněno v předešlé kapitole. Proto XPath řešení v této práci, má jako vstupní datovou hodnotu, **asociativní pole**.

Pole však nemá „primitivní“ strukturu, ale jde o definovanou strukturu, která je výstupem z funkce *to_array*.

Řešení je rozděleno na několik části, které na sebe navazují:

1. zpracování výrazu
2. procházení dat a uplatnění výrazu
3. aplikace podmínek
4. zpracování výstupního pole

5.1 Osy XPath

V úvodní kapitole bylo zmíněno u XPath, že obsahuje několik možnosti os. V této diplomové práci byla implementována pouze osa **::child**. Důvodem je hlavně složitost implementace, dalším důvodem je téma diplomové práce, kdy XPath je jen jedna z částí celého celku této práce.

V modulu třídy MySax lze použít např. tyto varianty XPath:

Tab. 2 Příklady implementovaných XPath výrazů

XPath výraz	Popis
/books/book/price	seznam hodnot cen
/books/book[2]/price	cena u druhé knihy
/books/book[price>2]	všechny knihy s cenou vyšší než 2
/books/book[price>150][price<500]	knihy s cenou vyšší než 150 a menší než 500
/books/book[description]	všechny knihy, které mají element 'description'
/books/book/.[1]	první kniha
/books/book/.[:last]	poslední kniha
/books/book/.[:first]	první kniha
/books/book[description=/popisek/i]	všechny kniha, které mají element 'description' a jeho hodnota splňuje regex /popisek/i
*/books/book/@**	vybere všechny elementy u knih (price, vat, …)
/books/book[@author=Vaclav Havel]	filtrace atributů

5.2 Proces zpracování

XPath řešení je implementováno v třídě *MySax*, reprezentováno funkci *xpath($path, $data = null).* Funkce má tedy dva vstupní parametry, výraz (string *$path)* a data (array $data). Na začátku funkce se ověřují podmínky na množinu dat (pokud je prázdna, návratová hodnota je prázdné pole), cestu (pokud je cesta zadána jako '/', bez dalšího zpracování se vrací celá struktura dat).

Následuje rozdělení XPath výrazu dle regulárního výrazu na jednotlivé úrovně dle dělícího znaku lomítka „/“.

```
$tokens = array_slice(preg_split('/(?<!=)\/(?![a-z-]*\])/', $path), 1);
do {
   //aplikace XPath viz kapitola 5.2.1
} while(1)
```

Rozdělení XPath výrazu pomocí regulárního výrazu

Dalším krokem je cyklus *while*, který se opakuje pořád dokola, dokud není zastaven. Zastavení cyklu znamená, že ve výrazu nezbyly žádné úrovně pro aplikaci na zbylá data.

5.2.1 Cyklus *while*

Před cyklem while, jsme získali pole jednotlivých úrovní.

```
$token = array_shift($tokens); //vybíráme první token (level)
```

Inicializace první úrovně do proměnné *$token*

V každém kroku cyklu while, je vybrána první úroveň a ta se bude aplikovat na data. Úroveň, která je zpracovávaná, je z tohoto pole odstraňována (každá úroveň je tedy aplikovaná jen jednou).

```
preg_match_all('/\[([^=]+=\/[^\/]+\/|[^\]]+)\]/', $token, $m))
```

Regulární výraz pro oddělení predikátů

Součástí každé úrovně (jak bylo zmíněno v úvodní kapitole o XPath) je název elementu, osa a predikáty. V této části z každé vybrané úrovně budou odděleny predikáty od názvu elementu, který je hledán. Nastavení podmínek je uloženo do proměnné *$conditions*. K zpracování podmínek se vrátíme na konci cyklu. Nyní se budou procházet data. Do proměnné *$matches* budeme ukládat výsledky. Na začátku je nastaveno na *false*.

Na základě toho, že názvy elementů jsou uloženy jako klíče, je aplikováno vyhledávání úrovně na nich. Procházení dat obsahuje dvě základní podmínky:

1. vyhledávaná úroveň se rovná *'/@*'*
2. hledaná úroveň existuje jako klíč v datech

První podmínka zachycuje výraz, kdy hledáme všechny potomky, včetně jejich obsahu bez jakéhokoliv omezení. Tedy aktuální data jsou uložena do proměnné *$matches*.

Druhá podmínka zpracuje obsah pole a ten uloží do proměnné *$matches*. Každý takový vyhovující klíč a jeho obsah je do výsledné proměnné uložen jako položka pole.

```
$matches[] = array(
  'trace' => array_merge($context['trace'], $ctext),
  'key' => $key,
  'item' => $item,
  '__content'=>$__content,
  '__attributes'=>$__attributes,
);
```

Ukázka obsahu pole *$matches*

Každý výsledek je rozdělen na jednotlivé složky:

trace	seznam předků
key	klíč (hledaný název elementu)
item	obsah prvku (potomci)
__content	datová část prvku
__attributes	atributy prvku

Na konci tedy zůstává v proměnné *$matches* uložený seznam výsledných prvku z dat. Nyní je nutno aplikovat predikáty na výsledná data. Ty, které podmínkám nevyhovují, jsou vyřazeny.

K rozhodnutí zda daný výsledek zůstane v poli či nikoliv rozhoduje funkce *matches*. Funkce aplikuje predikáty na obsah, počet prvků, atributy apod. Malý problém zde představovalo ošetření, zda predikát aplikovat na datovou část (_content) nebo na obsah (potomky).

```
$contexts = $matches;
if (empty($tokens)) {
  break;
}
```

Výsledky *$matches* jsou uloženy do *$contexts*

Po zredukování výsledků o nepotřebné elementy, nastavíme tento výsledek jako data pro další krok cyklu while.

Na konci je podmínka, která kontroluje, zda je nutno ještě něco „filtrovat“. Pokud je pole úrovní prázdné, cyklus while končí.

5.2.2 Zpracování výsledku

Po dokončení cyklu je nutné výsledné pole upravit, hlavně o nepotřebné informace jako seznam předků. Výsledkem XPath řešení je tedy seznam (pole) vyhovujících prvků, případně jejich potomků.

Výsledky jsou uloženy do pole zpět jako klíč a jeho obsah. Jedná se i o vizuální úpravu výsledného pole. V případě, že obsah prvku je pole o velikosti > 2, do obsahu pole uložíme celý jeho obsah daného prvku. V opačném případě uložíme jen *__content* složku, to hlavně z důvodu, kdy XPath výraz chce seznam potomků a je zbytečné vracet celý obsah výsledného pole, ale stačí jen obsah (__content).

Celý postup řešení XPath ve zdrojovém kódu lze vyjádřit pro lepší představu pseudokódem.

```
ZACATEK

 tokens := rozdel uroven dle regexp;
 dokud není false opakuj
     token := vyber prvni prvek z tokens;
     pokud obsahuje uroven predikat [] aplikuj regexp nad vytahnuti podminek
             token := token bez predikatu;
            conditions := podminky z regexp
     konec

     matches := prazdne pole;
     cyklus foreach (data)
          pokud token="@*"
               pridej do pole matches vsechny potomky data
          jinak jestlize token je klicem v data
              items := data[token];
             cyklus foreach (items)
                  pridej do matches obsah kazdeho prvku items
             konec
         konec
     konec


     pokud jsou nastaveny conditions
       filtred := prazdne pole;
       cyklus foreach (matches)
         pokud vyhovuje prvek tak ho prirad do filtred
       konec
       matches := filtred;
     konec
     data := matches;

     pokud je tokens prazdne
       vrat false ; //zastavime hlavni cyklus while
     konec
 konec

 //zpracovani vysledku
 r := prazdne pole;
 cyklus foreach (matches)
     pokud je match[item] je pole a pocet polozek match[item] > 2
         pridej do r match[item]
     jinak
         pridej do r match[item][__content]
      konec
 konec
 vrat r;

KONEC
```

Ukázka implementovaného řešení XPath pomocí pseudokódu

6 Validace XML struktury (XS schéma)

Účelem XML parseru je zpracovat (rozparsovat) danou XML strukturu.

Představme si případ, kdy se snažíme zpracovat obsáhlý XML soubor, zpracování zabere několik sekund. Na konci však zjistíme, že vstupním soubor neměl správnou (požadovanou strukturu), a dokonce nebyl validní vůči standardům. Je samozřejmě nemyslitelné a uživatelsky nepřijatelné, aby každý XML soubor byl nucený uživatel parseru pokaždé osobně kontrolovat.

Proto W3C zavedlo XML Schema Definition (XSD). Jedná se o definiční soubor, který znázorňuje, jak má být struktura tvořena a co má obsahovat.

XML schéma tedy definuje:

- místa v daném souboru, kde se mohou jednotlivé elementy vyskytovat
- definice atributů
- elementy a jejich sub-elementy
- pořadí elementů
- četnosti elementů
- povinnost (vyplněný/nulový)
- datové typy (xs:string, xs:float atd.)
- standartní hodnoty

6.1 Řešení v modulu XML parseru

Při řešení implementace, vznikl jeden problém. SAX XML Parser, který byl určen k parsování XML struktury do objektového modelu, nepodporuje žádnou nativní funkcí validaci dle XML schématu.

Bylo tedy nutné najít jiné řešení, které by splnilo tyto podmínky:

- nativní funkce (knihovna)
- možnost validace dle XML schématu
- validace probíhala bez načtení XML souboru do paměti (DOM)

Především poslední bod je ten nejdůležitější. Je logické, že funkce, která soubor validuje dle schématu, musí soubor přečíst. Tento způsob čtení souboru musí probíhat stejně jako u SAX (XML Parseru), tedy **sekvenčně**!

Takovýchto možnostmi, mezi kterými lze volit, není mnoho. Existují dvě řešení:

- *DOM*
- *XML Reader*

První možnost je rozebrána v úvodní kapitole. Tato metoda nesplňuje především nejdůležitější podmínku – nezpracuje XML soubor sekvenčně, tedy je DOM je pro tyto účely nepoužitelný.

Druhá možnost, *XML Reader* splňuje všechny předpoklady k použití pro validaci podle XS.

6.2 XML Reader

Tato knihovna využívá podobný princip čtení jako SAX, tedy sekvenčně (po částech). Rozdílem oproti SAX je v logice jeho čtení. Zatímco SAX staví na tzv. „push model“, zahrnuje aplikaci proudem událostí, které vyvolává. XML Reader, naopak využívá metodu „pull model“. Daný obsah zpracováváme, když si o to aplikace řekne. Tento model může aplikaci více zpřehlednit a zefektivnit práci se zpracovávaným obsahem, samozřejmě odstraňuje i určité omezení, které SAX vytvářel voláním jeho tří funkcí (start, obsah, konec).

Ukázka použití

```
$reader = new XMLReader();
$reader->open("test.xml");
```

Ukázka použití třídy XML Reader

Na začátku inicializujeme třídu *XMLReader* do proměnné, poté funkcí *open* připojujeme XML dokument, který chceme zpracovat (číst). Po těchto dvou řádcích je pak dokument připraven ke čtení. V případě že chceme začít číst, stačí použít funkci *read* a ta přečte část dokumentu.

V případě že funkce dojde na konec dokumentu, vrátí hodnotu *false*, v opačném případě vrací *true*. Čtení dokumentu lze tedy zpracovat např. v cyklu *while*.

```
while ($reader->read()){
  //zpracování načteného obsahu
}
```

Cyklus while pro přečtení celého dokumentu

Třída *XMLReader* obsahuje několik funkcí pro zpracovávání obsahu. Funkce, které podávají informace o právě přečteném obsahu (části) dokumentu. Např. pro zjišťování druhu uzlu a jeho název.

Dále např. pro textový obsah přečteného počátečního elementu lze použít funkci *readString*, která obsahuje jeho datovou část.

Další důležitou vlastností při zjišťování obsahu je konstanta *XMLReader::ELEMENT*. Tato konstanta definuje kód uzlu.

```
$books = array();
if ($reader->name == "book" && $reader->nodeType == XMLReader::ELEMENT){
  $books [] =  $reader->readString();
}
```

Parsování elementu ve funkce XML Reader

V takovém příkladu, je uložen obsah do seznamu *$books,* pokud je název elementu roven „book“.

Validace struktury podle XML schématu

Jak již bylo zmíněno v struktuře modulu. Validaci obsluhuje vytvořená třída *SchemaValidator*.

```
$xs = new SchemaValidator('test.xml',true);
if($xs->validateSchema('schema.xsd') == true){ echo 'xml je validni';}
else{echo 'xml obsahuje chyby';}
```

Použití třídy SchemaValidator

Na začátku je nutné předat konstruktoru XML soubor, který se má validovat. Druhý parametr je dobrovolný a nastavuje, zda se mají zobrazit i chyby, které validátor v XML dokumentu objeví. V případě, že parametr není nastaven je automaticky zvolena hodnota *false* (chyby nezobrazovat).

Po nastavení a inicializaci proměnné pro validátor je použita funkce *validateSchema*, která vyvolává samotnou validaci na dokumentu. Parametr, který se musí předat do funkce je povinný a jde o soubor *.xsd, jehož obsahem je schéma, podle kterého se bude validovat. Funkce vrací *true* pokud je soubor validní, v opačném případě vrací *false*. Pokud soubor obsahuje chyby a v nastavení konstruktoru je nastavena hodnota *true*, jsou ihned vypsány chyby v dokumentu.

```
function validateSchema($schema){
    $reader = new XMLReader();
    $reader->open($this->xml_file);
    $reader->setSchema($schema);
    while ($reader->read()){}
    if($reader->isValid() === false){
       return false;
    }
    else{  return true; }
}
```

Ukázka části zdrojového kódu funkce validateSchema

Z kódu je patrno, že je nutné po otevření XML dokumentu ke čtení připojit i dané schéma pomocí dostupné funkce *setSchema*. Funkce je schopna připojit i vzdálený soubor. Nemusíme tedy schéma mít na lokálním disku nebo na serveru, kde běží aplikace, ale může se předat URL souboru, kde se schéma nachází. Následuje samotné přečtení souboru za pomocí funkce *read*. Funkce, která vyhodnotí výsledek má název *isValid*. Příklad celé funkce *validateSchema*, která je jen část toho co je implementováno. Jedná se o pouze část kódu pro interpretaci funkčnosti validace.

Důležitou součásti je zobrazení a odchycení chyb, které se naskytují v XML dokumentu. Bylo by velice špatné, kdyby validace vracela pouze pravdu/nepravdu o validnosti dokumentu. Hledání chyb by zabralo hodně času. Proto pro XML je implementována knihovna *libxml*.

Chyby získáme pomocí její nativní funkce *libxml_get_errors()*, ta vrací pole chyb, které se v dokumentu naskytují. Pro zpracování chyb můžeme tedy použít např. cyklus *foreach*.

Zpracování chyb zajišťuje funkce *display_errors* v třídě *schemaValidator*.

```
function display_errors() {
    foreach (libxml_get_errors() as $error) {
       echo self::display_error($error).'<br />';
    }
    libxml_clear_errors();
 }
```

Zdrojový kód funkce displaty_errors

Z kódu lze vyčíst, že jednotlivou chybu ještě zpracovává další funkce – *display_error*, protože jednotlivé chyby lze ještě roztřídit podle jejich typu (*Notice*, *Warning*, *Error*). Jejich zpracování může zpracovat např. *switch*. Návratová hodnota jednotlivé chyby neobsahuje však pouze typ, obsahuje i její kód, zprávu chyby a řádek kde se nachází a název souboru (pokud je inicializována proměnná názvu zpracovávaného souboru).

```
switch ($error->level) {
    case LIBXML_ERR_WARNING:
        $return .= "<strong>Warning $error->code</strong>: ";
        break;
    case LIBXML_ERR_ERROR:
        $return .= "<strong>Error $error->code</strong>: ";
        break;
    case LIBXML_ERR_FATAL:
        $return .= "<strong>Fatal Error $error->code</strong>: ";
        break;
}
```

Rozdělení chyby podle typu

7 Porovnání využití paměti oproti DOM a SimpleXML

Předmětem této kapitoly je hodnocení vytvořeného modulu (SAX parser) a porovnat jej s již vytvořenými přístupy k XML – **DOM** a **SimpleXML**. Jedná se tedy o testování (porovnání) dvou základních kritérií, kterými jsou:

- náročnost jednotlivých přístupů na paměť
- rychlost zpracování

Důležitým faktorem je stanovení podmínky - testování paměti a zároveň rychlosti zpracování je nutno provést jako aplikaci *klient-server*. Server zde tedy představuje **Apache**.

První případ je tím nejdůležitějším kritériem při rozhodování programátora, který způsob použije vzhledem k velikosti dokumentu. Obě kritéria se částečně navzájem ovlivňují, čím větší nárok na paměť má jednotlivý přístup, tím je někdy i větší jeho doba zpracování. Je nutno vzít v potaz i fakt, že zpracování je ve většině případů jen malá část celé aplikace. Proto je nutné tuto část co nejvíce optimalizovat a vybrat vždy ten nejvýhodnější způsob.

Rychlost zpracování je také velice důležitým faktorem. V tomto případě se však neočekává tak veliký rozdíl, mezi jednotlivými druhy zpracování jako tomu je u prvního kritéria.

U obou kritérií bylo nutné na začátku zanalyzovat to, jakým způsobem je změřit. Jednotlivé způsoby a její vyhledávání se značně lišily, proto budou uvedeny až v následujících kapitolách každého kritéria.

Pro porovnávání (testování) bylo nutné také vybrat XML dokument (data), nad kterým byly prováděny testy. Nakonec byly čtyři dokumenty rozděleny podle velikosti obsahu dat:

1. velikost souboru cca 100kB
2. velikost souboru cca 1MB
3. velikost souboru cca 10MB
4. velikost souboru cca 45MB

Dokumenty měly však stejnou strukturu, která je uvedena v texu dále. Tyto dokumenty byly vytvořeny za pomocí PHP skriptu, který vygeneroval požadované množství. Na začátku bylo náhodně vytvořeno pole šesti knížek s určitými daty. Následovně byl nastaven počet opakování a skript vkládal tyto knížky jako sub-elementy „*book*“, které byly náhodně vybrány z tohoto pole.

Jako sub-elementy každé knížky *(<book>)*, bylo zvoleno 6 položek, každá knížka obsahuje i několik atributů pro možnost filtrování.

V dokumentech se objevuje následující struktura:

```
<catalog>
  <book id="bk101" version="66" instock="true" unit_instock="10">
        <author>Nejlepší autor</author>
        <title>Titulek knížky</title>
        <genre>Počítače</genre>
        <price>27.55</price>
        <publish_date>2011-01-31</publish_date>
        <description>Popis dané knížky</description>
  </book>
    ..
    ..
</catalog>
```

Příklad XML struktury

7.1 Paměťová náročnost (vytížení)

Tato kapitola je věnována měření paměťové náročnosti jednotlivých přístupu k XML. Na začátku bylo nutné analyzovat, jakým způsobem je možno paměť v PHP měřit a dostat nějaké hodnoty. Byl především kladen důraz na nalezení metod takových, které by v ideálním případě běžely jako součást skriptu a měřily paměť před a po vykonání zpracování XML. Důkladným zkoumáním bylo zjištěno, že takových možností není mnoho.

První možnosti, jak změřit paměť přímo při běhu skriptu, je nativní funkce ***memory_get_usage***, která se objevuje v PHP od verze 4. Tato funkce má jako návratovou hodnotu – aktuální počet alokované paměti PHP v bytech. Od verze 5.2 má i jeden parametr typu *boolean* $real_usage, jedná se o možnost vypsat reálnou velikost paměti alokovanou systémem (nastavenou na *true*).

```
echo memory_get_usage(); // 36640
  $a = str_repeat("Hello", 4242);
echo memory_get_usage() ; // 57960
```

Ukázka použití funkce memory_get_usage

7.1.1 Kompletní struktura

Jako první krok bylo nutné vyzkoušet využití paměti objektového přístupu, který byl vytvořen v této diplomové práci – třída ***MySax***.

```
$sax = new MySax();
$sax->set_filtration(array('/catalog/*'));
$xml = $sax->loadFromLocalFile('files/test_1kb.xml');
```

Použití třídy MySax

Jako vstupní soubor byl použit ten nejmenší (100kB). Nejmenší soubor, měl cca 250 sub-elementů knížek.

Tab. 3 Využití pamětí

	MySax	**SimpleXML**	**DOM**
Paměť na začátku	381.3984 KB	3.6693 MB	3.67 MB
Paměť na konci	3.6647 MB	3.67 MB	3.6712 MB
Celkem použito	3.2922 MB	720 Bytes	1.2031 KB

Již první měření ukazuje, že vytvořený objektový model SAX má obrovské nároky na paměť. Druhý krokem bylo změření zatížení nativních funkcí SimpleXML a DOM. Podle takového měření se může více posoudit zda SAX je opravdu náročnější na paměť eventuálně jsou SimpleXML a DOM na tom podobně, nebo jsou ještě horší.

V tabulce se objevují informace o využití paměti pro nativní funkce. Jak lze posoudit z předchozího měření třídy **MySax**, obě nativní třídy se jeví jako úspornější přístupy k paměti, nejlépe vychází použití SimpleXML. Rozdíly hodnot paměti na začátku jsou způsobeny spouštěním skriptu pro všechny přístupy na jednou. Takže se paměť postupně navyšuje, ale na měření využívané paměti to ovšem nemá žádný vliv. Následně byly provedeny testy na dalších testovacích souborech.

Tab. 4 Testování druhého souboru o velikosti 1MB

	MySax	SimpleXML	DOM
Paměť na začátku	381.3984 KB	380.8125 KB	382.1328 KB
Paměť na konci	32MB*	381.7109 KB	383.3359 KB
Celkem použito	32MB*	920 Bytes	1.2031 KB

** - přístup alokoval více než maximální povolenou hodnotu na serveru pro paměť (32MB), proto jako výsledek bylo do tabulky zaznamenáno maximum možné paměti.*

Z důvodu, že měření přístupu MySax skončilo chybou, byl MySax měřen zvlášť. Nativní knihovny SimpleXML a DOM byly opět měřeny současně. V dalších souborech bylo měření MySax zbytečné, protože by vykazovalo stejně hodnoty.

Tab. 5 Testování třetího souboru o velikosti 10MB

	SimpleXML	DOM
Paměť na začátku	380.8125 KB	382.1328 KB
Paměť na konci	381.7109 KB	383.3359 KB
Celkem použito	920 Bytes	1.2031 KB

Tab. 6 Testování třetího souboru o velikosti 45MB

	SimpleXML	DOM
Paměť na začátku	380.8125 KB	382.1328 KB
Paměť na konci	381.7109 KB	383.3359 KB
Celkem použito	920 Bytes	1.2031 KB

Z posledních dvou měření souborů o velikostech *10MB* a *45MB* pro nativní knihovny SimpleXML a DOM je zřetelné, že naměřené hodnoty se pro ně nemění. Zde vzniká podezření na špatné měření těchto knihoven za pomocí funkce *memory_get_usage*.

Bylo tedy nutné zanalyzovat tuto funkci a vyhledat možnosti měření pro nativní knihovny. Z analýzy této funkce v dokumentaci PHP a prohledání diskuzí a odborných článku vyplynulo, že funkce *memory_get_usage* není schopna měřit alokovanou paměť, kterou využívají knihovny. Funkce *memory_get_usage* tedy nezahrnuje paměť, kterou využívají knihovny v PHP.

7.1.2 Využití části dokumentu

V první podkapitole bylo měření prováděno na celém dokumentu, tato podkapitola bude naopak měřit náročnost na paměť při použití předfiltrace (MySax) nebo XPath (SimpleXML, DOM). Principem je ukázat výhodu přístupu SAX, který soubor nenahraje celý, ale pouze část kterou zvolíme. U nativních knihoven to bude naopak nahrání opět celého dokumentu do paměti a až při použití XPath, bude vytažena jen část dokumentu.

Z prvních měření je však patrné, že se opět projeví nemožnost změřit celkové zatížení nativních knihoven. Předpokladem pro toto měření však bude, že se projeví u nativních knihoven použití XPath. Všechny testy byly zaznamenány do jedné tabulky pro lepší přehlednost.

Tab. 7 Testování souborů s použití filtrace

Typ zpracování	Typ paměti	100kB	1MB	10MB	45MB
MySax	*Paměť na začátku*	386.7422 KB	386.7422 KB	386.7422 KB	386.7422 KB
	Paměť na konci	1.5976 MB	12.7726 MB	32 MB*	32 MB*
	Celkem použito	1.2199 MB	12.3949 MB	32 MB*	32 MB*
SimpleXML	*Paměť na začátku*	1.598 MB	12.773 MB	382.3594 KB	382.3594 KB
	Paměť na konci	1.5993 MB	12.7743 MB	1.828 MB	32 MB*
	Celkem použito	1.3359 KB	1.3359 KB	1.4546 MB	32 MB*
DOM	*Paměť na začátku*	1.5993 MB	12.7743 MB	1.8284 MB	382.3594 KB
	Paměť na konci	1.7482 MB	14.2481 MB	16.7884 MB	32 MB*
	Celkem použito	152.4141 KB	1.4737 MB	14.96 MB	32 MB*

** - přístup alokoval více než maximální povolenou hodnotu na serveru pro paměť (32MB), proto jako výsledek byl do tabulky zaznamenán maximum možné paměti.*

Toto měření se ukázalo jako daleko efektivnější. Hlavně v projevení změn alokovaných paměti u všech přístupu dle souborů, které se zpracovávaly. U všech přístupu se projevilo alokování paměti, která přesáhla nastavenou mez na serveru (32MB). MySax však přesáhl tuto paměť už v souboru předtím (10MB). I zde se však jeví přístup SimpleXML jako nejlepší. Naopak DOM, který musí využít další třídy pro XPath – *DOMXPath,* se projevil už velmi značně.

Všimněme si také faktu, kdy DOM má vždy výslednou využitou paměť menší o jeden soubor než ***MySax***. Např. když DOM alokuje 1,4 MB, MySax alokuje podobnou paměť už v přechozím souboru.

7.1.3 Jiné možnosti měření

Jelikož funkce *memory_get_usage* nesplnila přímé požadavky (její výsledky byly pro nativní knihovny zkreslené), bylo nutné vyzkoušet i jiný typ měření. Jediným použitelným řešením z další analýzy a hledání jiného typu měření, vyzněla knihovna ***Xdebug***.

Xdebug

Xdebug je profilovací nástroj z vývojového prostředí Zend Studia. Jedná se o nástroj, který sbírá data během vykonávání skriptu. Xdebug je především profilovací nástroj, který je určen k nalezení slabých míst v aplikaci a její optimalizaci. Podrobnějším studiem tohoto nástroje však byla zjištěna skutečnost, že dokáže měřit i rozdíl ve využití paměti mezi jednotlivými kroky.

Slabinou tohoto nástroje je fakt, že se nejedná o nativní knihovnu. Tedy takové testování vyžaduje vlastní server, popřípadě tvorbu virtuálního serveru. Xdebug je nutné potom nainstalovat a nakonfigurovat.

Jeho testování tak probíhalo na virtuálním serveru, který vytvořil program ***EasyPHP***. Xdebug trasování běhu skriptu je poté nutné nastavit v php.ini. V případě povoleného trasování stačí pustit skript, Xdebug ho automaticky celý odchytí a zaznamená každý krok do externího souboru. Problémem je však, že v případě obsáhlého skriptu, může takový soubor mít i 100MB. A jeho log je čistě textový, proto čitelnost a přehlednost v něm je vcelku špatná. V jednom řádku je každá volaná funkce, nebo alokována proměnná či její použití. [7]

```
TRACE START [2011-04-28 22:00:06]
0.6469    1950016       -> is_subaction('customers', '') /var/www/servman/pb_events.php:263
0.6471    1949992       -> echoloc('0', '5') /var/www/servman/pb_events.php:270
0.6472    1949992       -> is_subaction('suppliers', '') /var/www/servman/pb_events.php:272
0.6473    1949968       -> echoloc('0', '6') /var/www/servman/pb_events.php:279
0.6474    1949968       -> is_subaction('parts', 'add2inv') /var/www/servman/pb_events.php:281
0.6474    1949928       -> is_subaction('parts', 'showparts') var/www/servman/pb_events.php:288
0.6475    1949888       -> is_subaction('parts', '') /var/www/servman/pb_events.php:296
0.6481    1949744       -> echoloc('0', '10') /var/www/servman/pb_events.php:343
0.6482    1949736       -> is_subaction('invoices', '') /var/www/servman/pb_events.php:345
0.6499    1949472
TRACE END   [2011-04-28 22:00:06]
```

Ukázka trasovacího souboru.

Výsledné hodnoty paměti, byly téměř totožné jako v předchozím případě. Tedy nativní funkce tok paměti v trasovaném souboru, prakticky neovlivnily. Proto další testování s tímto nástrojem bylo zbytečné.

XHProf

Dalším nalezeným nástrojem pro měření paměti byl poměrně nový nástroj ***XHProf*** vyvinut společnosti *Facebook*. Jedná se také o knihovnu (open source řešení), kterou je nutné nainstalovat a speciálně nastavit. XHProf má však složitější použití, je nutné jeho volání a pohyb měření v skriptu vyvolat. Takové měření tudíž může být velice časově náročné, hlavně jeho úprava ve stávající aplikaci. [7]

Toto řešení přináší několik novinek, jako je zaznamenávání využití paměti a měření přesného procesorového času a především jeho rychlost. Další výhodou je zpracované HTML prostředí pro vyhodnocení výsledků, možnosti porovnávání jednotlivých výsledků mezi sebou, možnost zobrazení grafů závislostí jednotlivých funkcí apod. Jeho nevýhodou je však nemožnost použití na Windows serverech. Z časového důvodu proto nebylo toto řešení přesněji testováno, ale jeho význam hlavně pro vytvoření kvalitních statistik bude určitě veliký.

Jako poslední možnost měření lze uvést, měření bez severu **Apache**. Takovou možnost lze provést obdobným způsobem jako při tvorbě desktopových aplikací, tedy např. překlad jazyka v příkazové řádce. Takový způsob měření tedy dokáže změřit „čistý“ nárok na paměť, avšak tato možnost měření nesplňuje první podmínku (chybějící **Apache**).

7.2 Rychlost zpracování

Druhou kapitolou zhodnocení vytvořeného modulu MySax a srovnání s nativními knihovnami SimpleXML a DOM, je z hlediska rychlosti zpracování – „Jak dlouho trvá zpracování“, které se bude většinou uvádět v milisekundách nebo sekundách. I v této části bylo nutné zanalyzovat jak dobu zpracování změřit a opět v ideálním případě aby se jednalo o co nejmenší zásah do aplikace a serveru.

I v tomto případě bylo možné využít pro takové měření nativní funkci jazyka PHP – ***microtime***, Tato funkce má jako návratovou hodnotu UNIX čas v milisekundách. Její návratovou hodnotu lze i změnit jako typ *float*, kterou lze nastavit podle jediného parametru (opět typu *boolean* (zapnuto/vypnuto)).

Měření pomocí takové funkce je velice jednoduché, kdy opět stačí zavolat funkci *microtime* před volání skriptu a po něm. Výsledná doba skriptu je pak jejich rozdíl.

Tab. 8 Testování doby zpracování kompletní struktury (v sekundách)

Typ zpracování	100kB [sec]	1MB [sec]	10MB [sec]	45MB [sec]
MySax	0.0300	∞*	∞*	∞*
SimpleXML	0.0000	0.0300	0.2600	1.1800
DOM	0.0000	0.0300	0.2500	1.1900

** - přístup alokoval více než maximální povolenou hodnotu na serveru pro paměť (32MB), proto jako výsledek byla do tabulky zaznamenána nekonečná hodnota*

V porovnání s paměti třída MySax nemá tak špatné výsledky. Lze však změřit pouze první soubor, ostatní stejně jako u náročnosti na paměť je nemožné změřit dobu zpracování. Zde je malá ukázka, jak náročnost paměti ovlivňuje výsledek doby zpracování. Respektive to, co není schopno se alokovat v paměti, je nemožné změřit jeho dobu zpracování.

U nativních funkcí se na rozdíl od náročnosti na paměť generují značné rozdíly dle velikosti zpracovávaného dokumentu. Časové rozdíly mezi nativními funkcemi jsou minimální, i tak se SimpleXML jeví jako velice rychlý nástroj v době zpracování.

Tab. 9 Testování doby zpracování filtrované struktury (v sekundách)

Typ zpracování	100kB [sec]	1MB [sec]	10MB [sec]	45MB [sec]
MySax	0.0600	0.5500	∞*	∞*
SimpleXML	0.0000	0.0300	0.3200	∞*
DOM	0.0100	0.0200	0.2800	∞*

** - přístup alokoval více než maximální povolenou hodnotu na serveru pro paměť (32MB), proto jako výsledek byla do tabulky zaznamenána nekonečná hodnota*

I zde ve filtrovaném obsahu se projevuje závislost na paměti. V tomto případě však náročnost na dobu zpracování logicky narůstá, protože skripty musí provést množství vyhodnocení podmínek, které trvají nějakou dobu. Stále však mluvíme o době zpracování v řádech milisekund až sekund, což je téměř zanedbatelný čas v celkovém měřítku skriptu.

V nativních funkcích se projevila nemožnost v posledním souboru změřit dobu zpracování. Z pozorování lze odhadovat, že doba zpracování u posledního souboru by se pohybovala kolem 1 sekundy. Pokud ale nemáme soubor kam zpracovat, je vcelku zbytečné řešit jeho zpracování.

8 Závěr

Modul XML SAX Parseru se podařilo bez větších problémů naprogramovat dle požadovaných a stanovených bodů modulu. Cíl práce se tedy z hlediska samotného řešení parseru povedlo naplnit. Určitou komplikaci způsobila implementace XPath, která byla hlavně z hlediska doby zpracování velice náročná.

V závěrečné kapitole se objevily nečekané výsledky měření, které způsobily rozsáhlejší zkoumání samotné problematiky, než se v počátku zdálo nutné. Hlavně výsledky náročnosti na paměť byly od prvního měření velkým překvapením, které celou kapitolu porovnání značně rozšířily. Rozdíl v naměřených hodnotách paměti je hlavně způsoben nečekaným problémem, změřit jejich reálnou hodnotu. Kdy obě nativní knihovny vykazovaly prakticky neměnné údaje bez ohledu na velikosti zpracovávaného XML dokumentu. Proto měření paměti hlavně z hlediska nativních knihoven je nepřesné a to hlavně z důvodu, kdy žádné z měření nebylo schopno prokázat celkovou náročnost na paměť, kterou si vezme samotné zpracování nativních knihoven.

Faktem však zůstává, že vytvořený modul, který převedl XML dokument do objektu je příliš náročný na paměť. Celým problémem náročnosti je po dalším měření samotný vytvořený objekt. Při měření náročnosti paměti v jednotlivých krocích vyšlo najevo, že samotná inicializace třídy *Element*, která zastupuje každý element z XML struktury, zabírá bez dat skoro 1kB! Tato skutečnost poukazuje na velikou problematiku jazyka PHP a jeho zacházení s paměti u vytvořených objektů. Problémem PHP je i nemožnost nastavit typ jednotlivé proměnné jako tomu je např. v Javě nebo C#. Z tohoto faktu plyne i dojem, že alokace proměnné zabere až příliš zbytečné paměti.

Řešením, které by vedlo ke snížení celkového nároku na paměť vytvořeného SAX parseru by mohl být převod XML struktury přímo do pole bez vytvářeného objektu. PHP podle měření umí daleko efektivněji, co se týče paměti, pracovat s typem pole (*array*) než objektem (*object*).

Výhodou implementované třídy zůstává možnost daleko efektivněji pracovat s více strukturovaným dokumentem, který bude velmi obsáhlý. MySax dokáže jak nastavit požadované elementy a dokument, tak rozdělit a vybrat z nich jen požadovaný obsah. Před použitím je však třeba provést větší optimalizaci a implementovat možnost XML dokumentu do pole, který by měl zajistit menší nároky na paměť.

V poslední kapitole se ukázalo jako nejtěžší část nalezení správného a funkčního měření pro náročnost na paměť. Samotná analýza dostupných funkcí byla časově velice náročná. Závěrečná kapitola se však díky této nutnosti analyzovat všechny dostupné metody projevila velice kladně,

hlavně z hlediska rozšíření znalosti této problematiky, která bude velice cenná do další praxe v tomto jazyce.

Z měření tedy vyplynulo, že použitý programovací jazyk PHP není tak silně objektový a hlavně optimalizovaný pro paměť, jak se v počátku zdálo. I když od verze PHP 5 byl velmi výrazně posilněn objektový model, je tvorba a časová náročnost v jazyce Java nebo C# daleko rychlejší a efektivnější. Proto je nutné dbát na přednost PHP, kterou je především jednoduchost a podle ní je nutné aplikace v PHP vytvářet.

Výsledný modul je tedy lepší více zjednodušit (převádět XML strukturu do pole) a využít ho v případech, kdy nabízí možnosti, které ostatní knihovny nemají, tedy přefiltrovat daný XML obsah a vybrat jen určitou část.

Zdroje

1. *Wikipedia* [online]. Úvod do PHP. 2011. [cit. 2011-03-16]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/PHP>.
2. *Wikipedia* [online]. XML. 2011. [cit. 2011-03-16]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Extensible_Markup_Language>.
3. *Wikipedia* [online]. SAX. 2011. [cit. 2011-03-24]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Simple_API_for_XML>.
4. Bc. Jaroslav Kubáček, *Struktura XML.* [online]. 2011. [cit. 2011-04-23]. Dostupný z WWW: <http://xml.ic.cz/chapter1.html#d1e56>.
5. Software602, *XPath výrazy, funkce a aplikační události v 602XML* [online]. 2011 [cit. 2011-05-01]. Dostupný z WWW: <http://sp.602.cz/cms/index.php/602_dev/content/download/5037/29687/file/XPath%20v%C3%BDrazy,%20funkce%20a%20aplika%C4%8Dn%C3%AD%20ud%C3%A1losti.pdf>.
6. Jiří Kosek, *XML schémata* [online]. 2011. [cit. 2011-05-01]. Dostupný z WWW: <http://www.kosek.cz/xml/schema/>
7. Andi Gutmans, Stig Saether Baklen, Derick Rethans, *Mistrovství v PHP 5* : CP Books, 2005, 655 s. ISBN 80-251-0799-X

Seznam tabulek